The Blind Stone Tablet: Outsourcing Durability to Untrusted Parties

Peter Williams, Radu Sion [*]
Network Security and Applied Cryptography Lab
Stony Brook University
{petertw,sion}@cs.stonybrook.edu

Dennis Shasha [†]
Department of Computer Science
New York University
shasha@cs.nyu.edu

Abstract

We introduce a new paradigm for outsourcing the durability property of a multi-client transactional database to an untrusted service provider. Specifically, we enable untrusted service providers to support transaction serialization, backup and recovery for clients, with full data confidentiality and correctness. Moreover, providers learn nothing about transactions (except their size and timing), thus achieving read and write access pattern privacy.

We build a proof-of-concept implementation of this protocol for the MySQL database management system, achieving tens of transactions per second in a two-client scenario with full transaction privacy and guaranteed correctness. This shows the method is ready for production use, creating a novel class of secure database outsourcing models.

1 Introduction

Increasingly, data management is outsourced to third parties. This trend is driven by growth and advances in cheap, high-speed communication infrastructures as well as by the fact that the total cost of data management is 5–10 times higher than the initial acquisition costs [26].

Outsourcing has the potential to minimize client-side management overheads and benefit from a service provider's global expertise consolidation and bulk pricing. Providers such as Yahoo [10], Amazon [1–3], Google [4], Sun [9] and others – ranging from corporate-level services such as IBM Data Center Outsourcing Services [5] to personal level database hosting [6, 8] – are rushing to offer increasingly complex storage and computation services.

Yet, significant challenges lie in the path of a successful large-scale adoption. In business, health care and government frameworks, clients are reluctant to place sensitive data under the control of a remote, third-party provider, without practical assurances of *privacy* and *confidentiality*. Yet today's privacy guarantees of such services are at best declarative and often subject customers to unreasonable fine-print clauses – e.g., allowing the server operator (or malicious attackers gaining access to its systems) to use customer behavior and content for commercial, profiling, or governmental surveillance purposes [21, 22]. These services are thus fundamentally insecure and vulnerable to illicit behavior.

Existing research (discussed in Section 2) addresses several important outsourcing aspects, including direct data retrieval with access privacy, searches on encrypted data, and techniques for querying remotely-hosted encrypted structured data in a unified client model [24, 55]. These efforts are based on the assumption that, to achieve confidentiality, data will need to be encrypted before outsourcing to an untrusted provider. Once encrypted however, inherent limitations in the types of primitive operations that can be performed on encrypted data by untrusted hosts lead to fundamental query expressiveness constraints. Specifically, reasonably practical mechanisms exist only for simple selection and range queries or variants thereof.

In this paper, we introduce an orthogonal thesis spurred by the advent of cheap and fast disks and CPUs. We believe that in many deployments, individual data clients' systems are in a position to host and access local large data sets with little difficulty at no additional cost.

Holding data locally may appear to contradict the spirit of outsourcing, yet we argue that there is one essential data management aspect that cannot be hosted as such: *transaction processing for multiple concurrent clients*. In a world where client machines have become powerful enough to run local databases, we predict data management outsourcing markets will converge on supplying the transactional, network intensive services and availability assurances which are not trivially achievable locally.

[*] The authors are supported in part by the NSF through awards CNS-0627554, CNS-0716608, CNS-0708025 and IIS-0803197. The authors also wish to thank Xerox/Parc, Motorola Labs, IBM Research, the IBM Software Cryptography Group, CEWIT, and the Stony Brook Office of the Vice President for Research.

[†] Shasha's work has been partly supported by the U.S. National Science Foundation under grants IIS-0414763, DBI-0445666, N2010 IOB-0519985, N2010 DBI-0519984, DBI-0421604, and MCB-0209754. This support is greatly appreciated.

In this paper, we thus introduce a novel paradigm for solving the data management outsourcing desiderata: *a mechanism for collaborative transaction processing with durability guarantees supported by an untrusted service provider under assurances of confidentiality and access privacy*. In effect, we achieve the cost benefits of standard outsourcing techniques (durability, transaction processing, availability) while preserving the privacy guarantees of local data storage. This is accomplished by enabling data clients to collaboratively perform runtime transaction processing and interact through an untrusted service provider that offers durability and transaction serializability support.

In this context, data outsourcing becomes a setting in which all permanent data is hosted securely encrypted offsite, yet clients access it through their locally-run database effectively acting as a data cache. If local data is lost, it can be retrieved from the offsite repository. Inter-client interaction and transaction management is intermediated by the untrusted provider who also ensures durability by maintaining a client-encrypted and authenticated transaction log with full confidentiality.

In our model, each client maintains its own cache of (portions of) the database in client-local storage, allowing it to perform efficient reads with privacy, while relieving local system administrators of backup obligations. The key benefit thus becomes achieving data and transaction privacy while (1) avoiding the requirement for persistent client storage (clients are now allowed to fail or be wiped out at any time), and (2) avoiding the need to keep any single client-side machine online as a requirement for availability.

2 Related Work

Queries on Encrypted Data. The paradigm of providing a database as a service recently emerged [34] as a viable alternative, likely due in no small part to the dramatically increasing availability of fast, cheap networks. Given the global, networked, possibly hostile nature of the operation environments, security assurances are paramount.

Hacigumus et al.[33] propose a method to execute SQL queries over partly obfuscated outsourced data. The data is divided into secret partitions and queries over the original data can be rewritten in terms of the resulting partition identifiers; the server then performs queries over the partitions. The information leaked to the server is claimed to be 1-out-of-s where s is the partition size. This balances a trade-off between client-side and server-side processing, as a function of the data segment size. At one extreme, privacy is completely compromised (small segment sizes) but client processing is minimal. At the other extreme, a high level of privacy can be attained at the expense of the client processing the queries in their entirety after retrieving the entire dataset. Moreover, in [37] the authors explore optimal bucket sizes for certain range queries. Similarly, data partitioning is deployed in building "almost"-private indexes on attributes considered sensitive. An untrusted server is then able to execute "obfuscated range queries with minimal information leakage". An associated privacy-utility trade-off for the index is discussed. The main drawbacks of these solutions lies in their computational impracticality and inability to provide strong confidentiality.

Recently, Ge et al.[67] discuss executing aggregation queries with confidentiality on an untrusted server. Unfortunately, due to the use of extremely expensive homomorphisms (Paillier [58, 59]) this scheme leads to impractically large processing times for any reasonable security parameter choices (e.g., for 1024 bit of security, processing would take over 12 days *per query*). Current homomorphisms are not fast enough to be usable for practical data processing.

Avoiding the tradeoff between processing and computation time altogether, we allow efficient queries on encrypted data with full privacy by running queries on a client-side decrypted copy of the data. Thus, with no additional network transfer, and with a computational cost equivalent to running the query on an unencrypted database, we provide full query privacy. Since we run the queries on a copy of the database at the client side, so there is no need for expensive homomorphisms, either.

Query Correctness. In a publisher-subscriber model, Devanbu et al.deployed Merkle trees to authenticate data published at a third party's site [24], and then explored a general model for authenticating data structures [49, 50]. Hard-to-forge verification objects are provided by publishers to prove the authenticity and provenance of query results. In [55], mechanisms for efficient integrity and origin authentication for simple selection predicate query results are introduced. Different signature schemes (DSA, RSA, Merkle trees [53] and BGLS [17]) are explored as potential alternatives for data authentication primitives. Mykletun et al.[25] introduce *signature immutability* for aggregate signature schemes – the difficulty of computing new valid aggregated signatures from an existing set. Such a property is defeating a frequent querier that could eventually gather enough signatures data to answer other (un-posed) queries. The authors explore the applicability of signature-aggregation schemes for efficient data authentication and integrity of outsourced data. The considered query types are simple selection queries. Similarly, in [47], digital signature and aggregation and chaining mechanisms are deployed to authenticate simple selection and projection operators. While these are important to consider, nevertheless, their expressiveness is limited. A more comprehensive, query-independent approach is desirable. Moreover, the use of strong cryptography renders this approach less useful. Often simply transferring the data to the client side will be faster. In [60] *verification objects* VO are deployed to authenticate simple data retrieval in "edge computing" scenarios, where application logic and data is pushed to the edge of the network, with the aim of improving availability and scalability. Lack of trust in edge servers mandates validation for their results – achieved through verification objects. In [38] Merkle tree and cryptographic hashing constructs are deployed to authenticate the result of simple range queries in a publishing scenario in which data owners delegate the role of satisfying user queries to a third-party un-trusted publisher. Additionally, in [48] virtually identical mechanisms are deployed in database outsourcing scenarios. [23] proposes an approach for signing XML documents allowing untrusted servers to answer certain types of path and selection queries.

Sion has explored query correctness by considering the query expressiveness problem in [64] where a novel method for proofs of *actual* query execution in an outsourced database framework for *arbitrary* queries is proposed. The solution is based on a mechanism of runtime query "proofs" in a challenge - response protocol built around the *ringer* concept first introduced in [31]. For each batch of client queries, the server is "challenged" to provide a *proof of query execution* that offers assurance that the queries were actually executed with completeness, over their entire target data set. This proof is verified at the client site as a prerequisite to accepting the actual query results as accurate.

While many of these efforts have produced efficient query correctness verification mechanisms for specific kinds of queries, unique approaches are required for different queries. Moreover, it has proven difficult to *simultaneously* provide correctness and privacy with these techniques, since the construction of a query verification object typically requires knowledge of the query. To simultaneously ensure query correctness, query privacy, and database privacy for fully general queries (on relational or other types of databases), we instead verify only the *updates*. We guarantee that clients have correct views of the database, thus ensuring they also obtain correct query results.

Database Integrity and Audit Logs. In a different adversarial and deployment model, researchers have also proposed techniques for protecting critical DBMS structures against errors [46, 61]. These techniques deal with corruptions caused by software errors. In work on tamper proof audit logs by Snodgrass et al.[45, 63] introduces the idea of hashing transactional data with cryptographically strong one-way hash functions. This hash is periodically signed by a trusted external digital notary, and stored within the DBMS. A separate validator checks the database state against these signed hashes to detect any compromise of the audit log. If tampering is detected, a separate forensic analyzer springs into action, using other hashes that were computed during previous validation runs to pinpoint when the tampering occurred and roughly where in the database the data was tampered. The use of a notary prevents an adversary, even an auditor or a buggy DBMS, from silently corrupting the database.

This notion of a cryptographically protected log provides the framework for our solution. Rather than using an audit log to identify errors, however, we use an update log stored by an untrusted party as the authoritative version of the database; the cryptographic hash properties are used to prevent tampering by the untrusted party. Our log does not protect from software bugs in the DBMS, as audit logs traditionally do, since our logs store only a list of updates, not checksums on the contents.

Databases Replication. Database replication has been pursued as a means to improve query latency and throughput, and eliminate any single point of failure. Researchers have found efficient solutions guaranteeing various levels of consistency. In particular, [41], and later [13], build replicated databases on top of existing RDBMS's using a reliable group communication system with total message ordering and (in [13]) message logging.

We employ a similar model of operation, in which clients simulate transactions locally before distributing them. In place of the group communication system, however, we use a dedicated, untrusted provider, and provide cryptographic guarantees of security and privacy. This allows us to remove the availability and write durability requirements from the replication servers in previous work, assigning them to the untrusted provider, making client system administration much simpler.

Encrypted Storage. Encryption is one of the most common techniques used to protect the confidentiality of stored data. Several existing systems encrypt data before storing it on potentially vulnerable storage devices or network nodes. Blaze's CFS [14], TCFS [19], EFS [54], StegFS [52], and NCryptfs [69] are file systems that encrypt data before writing to stable storage. NCryptfs is implemented as a layered file system [35] and is capable of being used even over network file systems such as NFS. SFS [32] and BestCrypt [39] are device driver level encryption systems. Encryption file systems are designed to protect the data at rest, yet only partially solve the outsourcing problem. They do not allow for complex retrieval queries or client access privacy.

Integrity-Assured Storage. Tripwire [42, 43] is a user level tool that verifies file integrity at scheduled intervals of time. File systems such as I^3FS [40], GFS [27], and Checksummed NCryptfs [65] perform online real-time integrity verification. Venti [62] is an archival storage system that performs integrity assurance on read-only data. Mykletun et al.[56, 57] explore the applicability of signature-aggregation schemes to provide computation- and communication efficient data authentication and integrity of outsourced data.

In integrity-protected random-access storage, significant computational overhead is typically required to prevent rollback attacks, in which an adversary replaces a portion of the data with an older version. The party intending to detect such an attack may need a proof of the latest versions identifier of all stored data, for example. We avoid this overhead since each client keeps track of this information in a locally stored copy. We are then left with only the task of ensuring a consistent version sequencing on updates, which we provide using a hash chain.

Keyword Searches on Encrypted Data. Song et al.[66] propose a scheme for performing simple keyword search on encrypted data in a scenario where a mobile, bandwidth-restricted user wishes to store data on an untrusted server. The scheme requires the user to split the data into fixed-size words and perform encryption and other transformations. Drawbacks of this scheme include fixing the size of words, the complexities of encryption and search, the inability of this approach to support access pattern privacy, or retrieval correctness. Eu-Jin Goh [28] proposes to associate indexes with documents stored on a server. A document's index is a Bloom filter [15] containing a codeword for each unique word in the document. Chang and Mitzenmacher [20] propose a similar approach, where the index associated with documents consists of a string of bits of length equal to the total number of words used (dictionary size). Boneh et al.[16] proposed an alternative for senders to encrypt e-mails with recipients' public keys, and store this email on untrusted mail servers. Golle et al.[30] extend the above idea to conjunctive keyword searches on encrypted data. The scheme requires users to specify the exact positions where the search matches have to occur, and hence is impractical. Brinkman et al.[18] deploy secret splitting of polynomial expressions to search in encrypted XML.

We find we can efficiently (and trivially) achieve the spirit of this goal – running searches with full privacy on "outsourced" data – by adjusting the model to redefine "outsourced". Specifically, we provide a model that achieves the bulk of the cost savings of outsourcing while retaining the performance benefits of locally managed data.

3 Model

We provide details of the participants in this protocol, the required transaction semantics, and the cryptographic primitives employed.

Parties

Provider/Server. The provider owns durable storage, and would like to provide use of this storage for a fee. The provider, being hosted in a well-managed data center, also has high availability. We will investigate ways that clients can make use of these attributes.

Since the provider has different motivations than the clients, we assume an actively malicious provider. However, we do not try to prevent denial of service behavior from the provider. There are techniques beyond the scope of this paper, that can be employed to help clients detect denial of service behavior, such as attaching timestamps to messages to measure server latency.

Clients. In our model, the clients are a set of trusted parties who must run transactions on a shared database with full ACID guarantees. Since storage is cheap, each client has a local hard disk to use as working space; however, due to the fragile nature of hard disks, we do not assume this storage is permanent. Additionally, the clients would like to perform read queries as efficiently as possible without wasting network bandwidth or paying network latency costs. Each of the trusted parties would also like to be able to continue running transactions even when the others are offline, possibly making use of the provider's high availability.

The clients would like to take advantage of the durability of the provider's storage, but they do not trust the provider with the privacy or integrity of their data. Specifically, the provider should be prevented from observing any of the distributed database contents. We define a notion of consistency between the clients' database views to address integrity. It is not imperative that all clients see exactly the same data as the other clients at exactly the same time, however, they need to agree on the sequence of updates applied. We define $trace_{c,i}$ to be the series of the first i transactions applied by client c to its local database copy. Clients c and d are considered **i-trace consistent** if $trace_{c,i} = trace_{d,i}$.

In some scenarios the provider might be able to partition the set of clients, and maintain separate versions of the database for each partition. This partitioning attack has been examined in previous literature; if there are non-intercommunicating asynchronous clients, the best that can be guaranteed is fork consistency, first defined by Mazières and Shasha in [51]. Any adopted solution should guarantee that the data repository is fork consistent; that is, all clients *within each partition* agree on the repository history. This guarantee is not as weak it may appear to be on the surface, as once the provider has created a partition, the provider must block all future communication between partitioned clients, or else the partition will be immediately detected.

We assume that clients do not leak information through transaction timing and transaction size. Clients in real life may vary from this with only minimal loss of privacy, but we use a timing and size side-channel free model for illustration purposes.

The first part of this paper assumes a potentially malicious provider, but trusted clients. Section 6.1 relaxes this assumption to provide protection against not only a potentially malicious provider, but against malicious clients at the same time.

Transaction Semantics

We provide a general protocol that supports nearly any class of transaction. Transactions can be simple key-value pair updates, as in a block file system, or they can be full SQL transactions. Clients can even buffer many local updates over a long period of time, e.g. when the client is disconnected, and then send them as a single transaction. The only requirements for using this protocol are that the underlying transaction-generating system implement the following:

`RunAndCommitLocalTransaction`(Transaction T) applies transaction t to the local database and commits.

`DetectConflict`(TransactionHandle h, Transaction C) returns `true` if the external (program-visible) outcome of Transaction T_h would have been different had transaction C been issued before T_h. It does not matter whether the local database contents would be different; all that matters is whether the transaction issuer would have acted differently.

`Retry`(TransactionHandle h) rolls back all changes (in the local database, and any side-effects external to the database) for uncommited transaction T_h and reattempts the transaction.

`RollbackLocal`(TransactionHandle h) rolls back local database changes from uncommited transaction T_h.

We provide the following interface to the transaction-running system:

`DistributeTransaction`(Transaction T, TransactionHandle h) returns once transaction T has been successfully committed to the global database image. Implementations of this command will invoke the callbacks above.

Conflicts

In the protocols described later, multiple clients will simultaneously run transactions that ultimately end up in a different order than the clients see. Therefore, we define the notion of conflicts to indicate whether this re-ordering affects the client computation.

We say transactions a and b *conflict* if changing the order of these transactions affects the return value of one of the operations, or the client state that results from executing these operations. This definition allows us to provide the highest level of transactional isolation, serializability, as defined in the ANSI SQL-92 standard [12]. Client implementations may substitute a lower isolation level, such as one defined in [11], to improve performance by reducing the number of conflicts.

For example, if this system represents a block file system, clients may abort transactions that read the value of a block that is written in a prior, pending transaction. If this system represents a relational database, clients may use row-based or table-based conflict detection. Alternatively, some transactions could be implemented as PL/SQL procedures that avoid sending any values back to the initiator, avoiding any possibility of conflicts altogether!

For correctness, there must be no false negatives returned by the `DetectConflict` command (no client decides its transaction is conflict-free based on the transaction list, when it in fact is not). Of course, the definition of conflict-free depends on the particular implementation. For optimal performance, the number of false positives returned by the `DetectConflict` command must also be low.

We require several cryptographic primitives with all the associated semantic security [29] properties: (i) a secure, collision-free hash function which builds a distribution from its input that is indistinguishable from a uniform random distribution (we use the notation $h(x)$), (ii) an encryption function that generates unique ciphertexts over multiple encryptions of the same item, such that a computationally bounded adversary has no non-negligible advantage at determining whether a pair of encrypted items of the same length represent the same or distinct items, (iii) a pseudo random number generator whose output is indistinguishable from a uniform random distribution over the output space, and (iv) a recursive hash chain construction used to incrementally build a secure hash value over a sequence of items.

The first part of this paper assumes a potentially malicious provider, but trusted clients. All transactions are encrypted by a symmetric key shared by the set of clients, and kept secret from the provider. Message authentication prevents tampering, and the use of a versioning structure guarantees database cache consistency. A strawman protocol introduces solution by providing the security guarantees trivially using a global lock (Section 4). Our main result is a protocol providing these guarantees using an optimistic wait-free protocol (Section 5). We then describe several extensions to this protocol, including in Section 6.1 protection against not only a potentially malicious provider, but against malicious clients as well. Finally, our implementation shows how this protocol can be layered on top of existing SQL-based relation database management systems while obtaining practical performance overheads.

4 Strawman protocol: outsourced durability with a global lock

We start by illustrating the main concepts through a strawman protocol that allows multiple clients to serialize their transactions through an untrusted server – transaction atomicity being guaranteed through a single global lock. Naturally, in practice, global locking is not a viable option as it would constitute a significant bottleneck. Our main result, described in Section 5, is optimistic and lock-free.

An *encrypted transaction log* is the central data structure in all versions of this model. This log is the definitive representation of the database; the protocols described here simply allow clients to append to this log in a safe, parallel manner while preventing the potentially malicious provider from interfering with operations on the log.

At a high level, in this strawman protocol, clients maintain their own copy of the database in local temporary storage. They perform operations on this copy and keep it synchronized with other clients. Clients that go offline and come back online later, obtain a client-signed replay log from the untrusted server in charge of maintaining the log.

4.1 Transaction Protocol: the lock-based `DistributeTransaction`

Informally, to run a transaction a client (1) "reserves" a slot in the permanent transaction log, (2) waits for the log to "solidify" up to its assigned slot, (3) runs the transaction, (4) "commits" that slot by sending out a description of the transaction to the untrusted server. The untrusted server then archives and distributes this encrypted transaction.

1. The client issues a "request slot" slot reservation command to server, along with a number l representing the last slot the client knows about (has seen updates for). The server assigns the next available transaction slot s to the client. The server sends back to the client this slot number s, with a list of all commits since the last update T_l received by the client, $T_{l+1} \ldots Tj$. Note that $j < s - 1$ if there are clients reserving slots that have not yet committed at the instant the server issues the response to this message.

2. The client waits until all transactions $T_{j+1} \ldots T_{s-1}$ in slots before its assigned slot have committed. The client receives the updates from the server as they come in. After verifying checksums and authentication tokens (hash chain and signatures; see below) on each update, the client applies them to its local database copy (using `RunAndCommitLocalTransaction`) in order.

3. Once the client has received T_{s-1}, it has in effect obtained a global lock, since all other clients are now waiting for it to perform a transaction. The client now runs its own transaction on its local copy of the database, recording the sequence of updates.

4. The client commits (relinquishing the lock) by sending a complete encrypted description of the transaction updates T_s back to the server (which will relay it back to the other clients).

Finally, each client c applies transaction T_i (using `RunAndCommitLocalTransaction`) to its database once all the following conditions hold: (i) the contents of T_i have a valid signature from a valid client (using client-shared symmetric key K), (ii) the client has applied transactions $T_1 \ldots T_{i-1}$, and (iii) the hash chain link T_{i-1}.hashchain matches the client's own computation of link HC_{i-1}.

Each transaction T_i is encrypted and signed using a symmetric key K shared by all clients. It contains the following fields: **desc=** a transaction description, e.g., a sequence of SQL statements; **hashchain=** the signed hash chain link HC_{i-1}, verifying the sequence of transactions $T_1 \ldots T_{i-1}$. HC_i is calculated as $H_k(HC_{i-1} || T_{i-1}.desc)$, with $HC_0 = H_k(\emptyset)$.

```
hashchain := $H_k(\emptyset)$
hashchain_pos := 0
k := ClientSharedKey
```

Global variables

```
Result: Retrieves and runs the next waiting transaction
t := server_getNextTransaction()
if !verifySignature(t) then
    return $\perp$
end
T := decrypt(t)
if T.hashchain != hashchain then
    return $\perp$
end
RunAndCommitLocaltransaction(T.desc)
hashchain := $H_k$(hashchain || T.desc)
hashchain_pos ++
```

Procedure `GetIncomingTransaction`

```
Result: Initiates a transaction
s := server_requestSlot()
while hashchain_pos < s-1 do
    GetIncomingTransaction()
end
RunAndCommitLocalTransaction(d)
T := new Transaction
T.desc := d
T.hashchain := hashchain
server_commit(Enc(k,T))
```

Procedure `DistributeTransaction(`*d*`)`

4.2 Correctness

We now show that this protocol is correct and offers fork consistency: all clients are trace consistent (their transaction traces are identical) as long as the server has not partitioned the clients. If the server *has* partitioned the clients, then all clients within a partition will be trace consistent.

Theorem 1. *If client* c *applies an update* T_i *of client* d*, then clients* c *and* d *are* i*-trace consistent.*

Proof. Assume that client c has applied T_i from client d, but suppose that $trace_{c,i}$ differs from $trace_{d,i}$. W.l.o.g. assume that at position a, $trace_{c,i}$ contains transaction T_a while $trace_{d,i}$ has a different value T'_a. Client c's computation of HC_a therefore differs from client d's computation of HC_a, since it involves a collision-free hash function. Additionally, the collision-free property guarantees that any later link in these hash chains will also differ, and client c's computation of HC_{i-1} differs from client d's computation of HC_{i-1}. Since client c has applied T_i, it had to have successfully verified that T_i did indeed originate from client d. However, as a pre-condition to client c applying T_i, the hash chain link T_{i-1}.hashchain, which is client d's computation of HC_{i-1}, would have to match client c's own calculation of HC_{i-1}, giving us a contradiction. □

4.3 Privacy

Without knowledge of the shared symmetric key K, the server is unable to obtain any information from the encrypted transaction descriptions, aside from timing and size. Transaction slot requests contain no additional information.

5 Lock-free outsourced serialization and durability

The obvious disadvantage to the above protocol is that it requires a global lock, restricting transaction processing as only one client may be active at a time. We now remove all locking from the protocol described above, and replace it with an optimistic conflict-detection mechanism. This allows clients to run transactions simultaneously, but adds the requirement that transactions are rolled back and reissued in the case of conflicts.

At an overview level, this protocol works as follows. Clients first issue an (encrypted) notification of their pending transaction, relayed to the other clients through the untrusted server. This contains enough information to allow other clients to determine whether it might cause a conflict with their own pending transactions. After this notification ("pre-commit"), clients then check to see if their pending transaction might conflict with any transactions scheduled to run before theirs. If not, they issue the commit; otherwise they retry with a new request. As in the previous protocol, clients maintain a transaction hash chain to guarantee consistency for misbehaving servers.

5.1 Transaction Protocol: the lock-free `DistributeTransaction`

In this solution, running a transaction entails the following steps, outlined in figure 1:

1. The client simulates the intended transaction on its local database copy, then undoes this transaction on its own database copy. (Issuing the `RollbackLocal` client command defined in the Model section). It will properly apply the transaction only once it has applied the pending transactions first.

2. Once ready to commit, the client issues the "Request slot" command to the server, attaching an encrypted pre-commit transaction description P of its intended transaction, and the slot number l which is the latest the client knows about.

3. The server allocates a slot s, and sends back a list of all new pre-commit descriptions $P_l \dots P_{s-1}$ up to s. The server may choose to also send any previously committed transactions that the client hasn't seen yet at this point (e.g., this is the case if the client just joined or has been offline for a while).

4. The client verifies the signatures on each pre-commit, and checks whether its transaction conflicts with these pre-committed transactions (conflict semantics were discussed in Section 3). E.g., a conflict occurs with pre-commit P_j, $l < j < s$ if the *external* state would be different depending on which of P_j or P_s is run first (the `DetectConflict` command identifies these conflicts). If there are no conflicts, the client commits by sending a final encrypted transaction commit C_s. If there are conflicts, the client still sends the commit C_s, but sets its abort flag first (see below). In the case of a conflict, the client also rolls back the external effects of running the transaction locally (using the `Retry` command).

5. The server commits by logging the encrypted transaction to permanent storage. It informs all other clients about the new transaction by sending the final encrypted transaction C_s.

The pre-commit transaction description P_i contains the following information, encrypted and signed with the symmetric key K shared by all clients: **desc=** a transaction description, e.g., a sequence of SQL statements

The final encrypted, signed transaction C_i contains the following information: **commit=** a single bit indicating whether this is a commit or an abort; **prehashchain=** hash chain link $HCPre_i$, verifying the sequence of pre-commits $P_1 \dots P_i$.

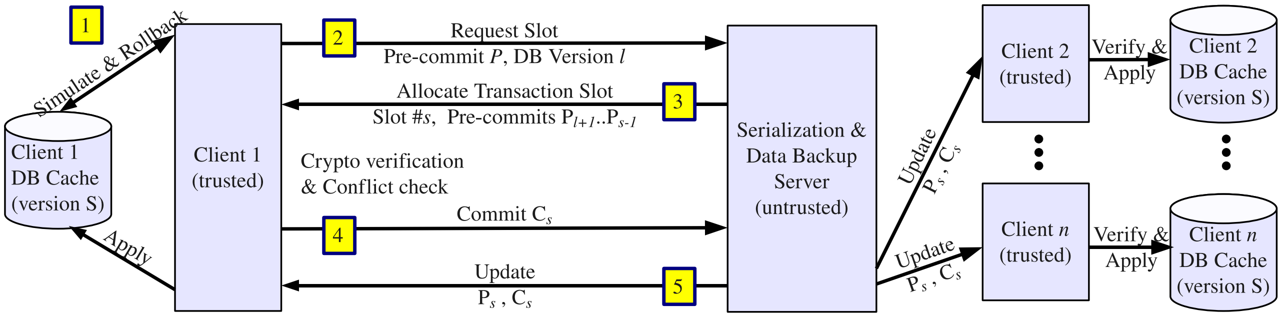

Figure 1. Performing an update: overview of `DistributeTransaction` execution, providing lock-free outsourced serialization and durability

Note that when issuing commit i, the client has seen all *pre-commits* up through i, because the precommits up to i are returned when the client is assigned slot i. However, the client may not have yet seen all *commits* up to i when issuing this commit C_i.

Client c applies transaction i (invoking client command `RunAndCommitLocalTransaction`, originating from client d, once the following conditions hold: (i) the contents of P_i and C_i have a valid signature from a valid client (using client-shared symmetric key K), (ii) C_i.commit indicates this is a committed transaction (not aborted), (iii) the client has applied transactions $1 \ldots i-1$, and (iv) the hash chain link C_{i-1}.pre-hashchain matches the client's own computation of link HC_{i-1} A pseudocode approximation of these steps is included for reference.

```
prehashchain[] := [$H_k(\emptyset)$]
links := 0
```

Global variables for subsequent pseudocode

```
Result: Verifies and runs the transaction, reported by
        the server
if !verifySignature(p) or !verifySignature(c) then
    return $\perp$
end
P := decrypt(p)
C := decrypt(c)
if prehashchain[id] != T.prehashchain then
    return $\perp$
end
if id == links+1 then
    prehashchain.append($H_k$(prehashchain[links] ||
    T.desc))
    links++
end
if  C.commit then
    RunAndCommitLocaltransaction(T.desc)
end
```

Procedure `IncomingTransaction` (*p,c,id*)

We note that this protocol is *wait-free*: if a client reserves a slot, sends its pre-commit but never completes, other clients can still perform transactions as long as they never access uncommitted data.

5.2 Correctness

We can again prove "fork consistency" for clients, as in the locking version. The difference here is that clients must agree on both C_i and P_i (the commit/abort status). Fortunately, only the client that issued C_i is allowed to issue P_i, so the proof follows.

Theorem 2. *If client* c *ever applies an update* T_i *of client* d*, clients* c *and* d *are* i*-trace consistent.*

Proof. Assume that client c has applied transaction i from client d, but suppose that $trace_{c,i}$ differs from $trace_{d,i}$ at position a. There are two possible ways in which $trace_{c,i}$ could differ from $trace_{d,i}$. Either the transaction description P_a.desc differs between clients, or the commit status C_a.commit differs. If the transaction description differs, then this reduces to logic in Thereom 1: there is a hash function collision. Now consider the second case, that the commit status C_a.commit differs between two clients. The C_a.commit is obtained in a signed message from the client originating transaction a; therefore, since we assume clients follow the protocol, the originating client has signed two separate commit statements for the same transaction. Thus, the client has misbehaved, which contradicts our assumption of correct client behavior. □

In summary, we guarantee fork consistency in the pres-

```
Result: Runs a transaction initiated locally
RollbackLocal(h)
P := new Pretransaction
P.desc := T
s,pending :=
server_requestSlot($Enc_K$(P),prehashchain.length)
conflict := false
foreach p,id ∈ pending do
    if !verifySignature(p) then
        return ⊥
    end
    $p_{id}$ := decrypt(p)
    prehashchain.append($H_k$(prehashchain[links] ||
    T.desc)
    links++
    if DetectConflicts(h, $p_{id}$.desc) then
        conflict := true
    end
end
c := new Commit
c.commit := !conflict
c.prehashchain := prehashchain
server_commit($Enc_K$(c))
if conflict then
    return Retry(h)
end
```

Procedure `DistributeTransaction(`*T, h*`)`

ence of a potentially malicious server using a shared hash chain. If two views of the ordered list of transactions ever differ between two clients, this will result in the shared hash chain diverging at the point of the inconsistency. This proof assumes correct clients. Section 6.1 shows how to prevent misbehaving clients from causing inconsistencies.

5.3 Privacy

In the absence of timing attacks, content, read/write access patterns and transaction dependencies are hidden from the server. This follows by construction as the contents of all messages are encrypted.

5.4 Forward Progress

Clients running this protocol will never deadlock, as long as they are not blocking for any external resources, since there is always the ability to make progress. This is evident since each transaction depends only on the transactions preceding it; the serialization numbers ensure there can never be any circular dependencies. At any point in time, there is always at least one transaction, at the front of the list, without any pending transactions to interfere.

For clients that are waiting on external resources, we can guarantee they will avoid deadlock as long as they (directly or indirectly) hold only external resources unneeded by prior, pending transactions. That is, our serialization technique assigns all transactions an ordering that makes it easy to prevent external resource deadlock as well.

"Livelock" and starvation are relevant concerns, however, and their applicability will depend on particular implementations. If a client detects it is being continually starved (i.e., there are always pending conflicting transactions), one solution is to block while waiting for the transaction chain to solidify up to a particular slot, since forward progress is guaranteed for the pending transactions. Conversely, a client must never block for a transaction past its slot. This forward independence prevents deadlock, and it also gives flexibility to client implementations; if a client needs a lock on a set of records, for example, it can request a transaction slot, then block until all transactions prior to the slot are committed. The client can then perform reads and writes with the equivalent of a lock. Meanwhile, other clients can prepare transactions to run in the future, under the restriction that their transactions do not conflict with the pending transaction. Random backoff is an alternate solution. This will let clients escape from livelock, but requires active participation of both the starved and the healthy parties.

5.5 Client Initialization

When a new client comes online, there may be a long list of updates it must apply from the transaction log. To the time required for client initialization, we recommend clients create and sign periodic database snapshots, up to any particular transaction number. The untrusted server hosts these database snapshots, which can be used by clients to recover a particular version of the database. The remaining uncovered portion of the log is then used to get fully up to date.

Database snapshots can similarly be used to reduce the storage requirements of the untrusted provider. Once a snapshot of version i exists, the transaction log entries from 0 to i can be discarded. Using database snapshots in combination with the transaction log to allow faster recovery is a traditional DBSM method in common use.

5.6 Privilege Revocation

Depending on the particular implementation, it may also be useful for the decision to revoke access from a client to be made externally, by a trusted party/system administrator, or internally, by a quorum of clients. Once the remaining clients agree to revoke access, they choose a new symmetric encryption key. Additionally, clients agree on a slot at which the client is considered terminated. This termination point can be determined by a system administrator, or by a quorum of clients. Modifications to the database after this slot by the terminated client are all rejected (ignored) by everyone else. Incomplete transactions are easy to discard once the remaining clients can come to an agreement about which are incomplete.

After revocation, the only abilities retained by the terminated client from its former access is read access on the database for transactions before the termination point, and potentially the ability to cause denial of service. To remove the advantage the revoked client has in performing a denial of service attack on the database, the service provider should be notified. This operation is not strictly necessary, since we still provide correctness even when the revoked client and the storage provider are colluding.

6 Protocol extensions

We now briefly describe several protocol extensions.

6.1 Malicious clients

We describe here a simple extension to the lock-free protocol that allows us to prevent a malicious client from bringing the rest of the system into an inconsistent state. Preventing data overwriting by a malicious client is a separate concern, which is most appropriately addressed at the database access control level.

To detect malicious client behavior before the system becomes inconsistent, we require two modifications. First, we extend the transaction integrity checks to include non-repudiation, so that each message is traced back to the issuing client. Specifically, we can replace the symmetric MAC function with a public key signing system. This prevents one client from impersonating another, and is required both to enforce the access control policy, and to gain accountability if incorrect behavior is detected.

The pre-commit hash chain already ensures that all clients agree on the pending transaction; to additionally protect from misbehaving clients colluding with the server, we simply need to ensure that all clients also agree on the commit/abort status of each transaction.

Since our protocol is lock-free and wait-free, clients will not necessarily know the commit/abort status of every transaction prior to their own as they issue a commit. Therefore, we employ a delayed-verification mechanism: as part of commit C_i, the client includes the following items: (i) **commit-hashchain-position=** the position $j < i$ of the last element in the chain this client can build (i.e., this client has received $C_1 \ldots C_j$ but has not yet received C_{j+1}), and (ii) **commit-hashchain=** The value of hash chain element j.

Clients must also cache some prior hash links in order to verify the link included with commit C_i, since $C_i.\text{commit-hashchain-position}$ might be less than $i - 1$. This cache size can be configured, and for most transaction scenarios it is likely safe to keep only a few entries.

Note this provides a slightly weaker guarantee concerning the status of commit/abort. In the presence of a malicious client colluding with the server, clients c and d will not necessarily be k-trace consistent. However, after both c and d have applied the inconsistent transaction j, the next update issued by client d (which must contain a commit-hashchain-position $\geq j$) will reveal the inconsistency to client c.

Theorem 3. *If clients c and d behave correctly, and client c has applied transaction j and issued a transaction for slot $k > j$, and client d applies the update at k from client c, then c and d are j-trace consistent.*

Proof. Assume that client d has applied update k from client c. Thus, client d's computation of $HC(k)$ agrees with C_k.pre-hashchain, and the contents of the transactions are consistent; the inconsistency is therefore in the commit/abort status of transaction j. Since client c applied transaction j before issuing update k, $C_k.\text{commit-hashchain-position} \geq j$. Similarly, since client d has applied transaction i, client d verifies that $C_k.\text{commit-hashchain}$ matches d's own computation of that chain link. However, since these two chain links have different values as inputs (one indicating that transaction j committed, and one indicating it did not), there is a hash function collision. □

In summary, we prevent malicious clients from causing inconsistency using an access control policy framework to limit data damage, non-repudiability of messages to prevent cross-client impersonation, and an additional hash chain to ensure clients agree on transaction commit/abort status.

6.2 Lowering transaction latency

We can reduce the number of network round trips required for a transaction commit from two to one by eliminating the commit messages C_i, as long as all clients have identical conflict detection logic. If we add another field to P_i indicating the last transaction the submitter has applied to its local database copy before this attempted transaction, other clients have enough information to determine the conflict status of this transaction! Thus, the commit flag in C_i.commit is redundant, at the expense of performing the conflict detection across all clients instead of just one.

The hash chain confirmation C_i.pre-hashchain will need to be placed in P_i, while adding another field C_i.pre-hashchain-location, since the submitter does not have enough information to build the entire pre-commit hash chain at this point. Thus, inconsistency checking is delayed slightly, and it requires longer to detect malicious behavior. Specifically, an inconsistency introduced due to server misbehavior will be detected only once a client that has applied the inconsistent transaction has sent a later update out to other clients who have seen a different transaction in that slot. This guarantee about detecting server misbehavior is similar to the guarantee about detecting client misbehavior in Section 6.1.

In conclusion, a simple modification to this protocol improves transaction latency by eliminating the commit message, at the expense of slightly more client computation time and slightly weaker consistency guarantees.

6.3 Large databases

So far we have not discussed the issue of local space limitations. We assumed up to now that clients can fit the entire database in local (volatile) storage, so that they can run queries without any help from other parties. If this is not the case, protocol extensions are necessary to allow clients to run queries. We discuss several mechanisms below.

On-demand data. Clients can use a separate query protocol to pull pieces of recent database snapshots from other clients, or authenticated database snapshots directly from the provider. This work-around has two drawbacks: first, access pattern privacy is forfeited if clients query the provider directly for only portions of the database. Second, performance suffers since sections of the database must travel the network multiple times.

Large object references. If clients can fit the database except for a set of large objects, the client can fetch these encrypted objects from the provider using a separate protocol. Access pattern privacy to these objects is lost, however access pattern privacy to the database indexes is preserved. In practice, privacy to the indexes is the most important part of privacy, since the indexes are subject to the largest semantic leaks, since positions within the index are correlated to contents of the database itself. Thus, this technique offers a useful privacy/storage tradeoff.

Performance will be mostly unharmed by the large object references work-around, as long as the bulk of the transaction processing work concerns only index data. The client's available storage is well suited for caching some of the most popular items, so most objects will traverse the network only a small number of times under most usage patterns. This technique suggest a modification to the transaction protocol to improve performance, to surpass in some scenarios even the performance of the original model: for operations on sets of large objects, clients announce the writes in the transaction log, but include only a hash of the large object content. This way, since the large object content is excluded from the transaction log, clients will not download the large objects at all, unless they are specifically needed for a query.

The only modification to the transaction protocol necessary to perform this operation is that clients include the object ID and a hash of the object content, as the content in the transaction field. Thus, the link (with a checksum and version) to the object is the stored content in the log and databases, and the object itself is an external entity. Clients treat the external object as if the updates occur when the link occurs in the transaction log, with naturally following semantics for transaction aborts and so forth.

Large object references with PIR. A Private Information Retrieval algorithm can be used to retrieve these large objects without revealing which objects are being retrieved, as long as the PIR algorithm does not reveal the size of the object, or the size of the object is not unique enough to allow an access pattern privacy-defeating correlation between the objects. The advantage of the overall scheme in this context is that access pattern privacy is preserved efficiently for the bulk of the computation; when large objects are retrieved (presumably less frequently), the more expensive PIR (such as [68]) is employed to preserve access pattern privacy.

The key to the practicality of all of these alternatives is that all the database indexes required to satisfy a particular query can fit simultaneously on a client, and that the client has enough working memory to perform the necessary joins efficiently. In practice we believe many databases are of a suitable form, with the bulk of the space consumed by large objects that do not need to be retrieved to compute joins.

6.4 Expiring Slots

There is a potential denial of service behavior if a client reserves a transaction slot but never commits; no transactions past this slot will be applied. A potential solution is using "mortal locks" that expire.

The following scenario outlines a method by which clients can safely delete expired locks: a pre-transaction reserved slot is only useful for a predetermined amount of time, specified by the client as it reserves its slot (or set as parameter). Clients timestamp the pre-transaction.

If this time has expired and the transaction is still in the pre-transaction phase, any client is now allowed to abort this transaction. The client desiring to abort the transaction simply issues to the storage provider an abort entry for this slot, which is then appended to the transaction log. The provider ensures that only the abort or the commit are appended to the log. The provider decides race conditions, and one of the operations will fail if both the abort and commit are issued. Clients can guarantee consistent provider behavior in fulfilling these new obligations, by using transaction hash chains as before: if the (untrusted) provider ever accepts both the abort message and the commit for a particular transaction, it will be obvious from the conflicting hash chains once the provider sends updates out (thus maintaining fork consistency).

6.5 Vague Pre-commit

We describe an extension here that allows clients to issue vague pre-commits, determining the final transaction contents only after their request slot has been reserved. This technique allows improved performance in certain conflict-heavy scenarios, by giving clients the flexibility to choose their transaction *after* they are informed of current operations. Clients might choose to modify their transaction to avoid conflicts, as an example.

In the above described lock-free protocol, clients submit a pre-commit indicating their pending transaction, then issue a commit or abort on this transaction after checking for conflicts. With an extension we can allow the commit version of the transaction to differ from the pre-commit version, adding the following field to the commit message C_i:
description= The actual transaction to run (instead of the P_i.description.

The only requirement added is that C_i.description be a "subset" of P_i.description. That is, any conflict that the final commit C_i might cause with future transactions would also be caused by the pre-commit P_i.description. With this requirement enforced, all client behavior is identical to what it would have been if the original P_i.description was C_i.description, with the exception that there might be more aborts than otherwise. This requirement is thus sufficient to ensure consistency when clients are honest. In the malicious client scenario, it is additionally required that all clients can determine whether any commit C_i.description is indeed a subset of the pre-commit P_i.description, as they don't trust the issuer to make that declaration.

7 Implementation and Experiments

Strawman Implementation (ODP). We built a proof-of-concept strawman implementation of the Outsourced Durability Protocol (ODP) using different components in Java, Python and C. The implementation handles SQL queries and relational data sets and runs on top of MySQL 5.0 [7], though with minor modifications we can support other RDBMS's. The protocol enables parties with low uptime to keep databases synchronized through a single, untrusted third party that has high uptime. Thus we allow safe outsourcing of both data backups and data synchronization through an untrusted provider.

In our particular setup we aimed towards simplicity rather than performance, giving each client application its own connection to a single database in the client's cluster. These connections are filtered through a proxy, which captures queries for our protocol to ensure proper propagation and conflict avoidance. Each cluster runs a single process that communicates with an untrusted service provider conduit through symmetric XML-RPC channels.

To filter queries we use MySQL Proxy [44], an open source scriptable tool built by the creators of MySQL, allowing capture and insertion of SQL queries and database responses. This simple setup shows that we can deploy quickly on existing systems while obtaining reasonable performance; a tailored solution would improve overhead by eliminating the numerous process forks, file writes, and TCP connections initializations in every transaction in the simple strawman implementation.

Strawman: Throughput Experiments. We performed experiments aimed at understanding the throughput behavior of our mechanisms. Given their network-dependent nature we focused on understanding how network characteristics impact performance.

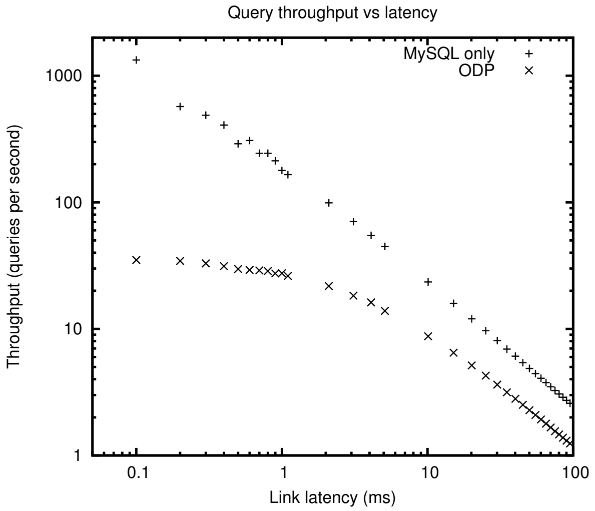

Figure 2. Query throughput in transactions per second vs. link latency, with log scale axes. Both MySQL and ODP quickly converge to a relationship inversely proportional to link latency.

The experimental setup consists of an (untrusted) "server" and several "clients" connected directly through a 1Gbps router. The server is a Dell PowerEdge 2850 running Centos 4.1 with 4 Dual core Xeons and 4GB RAM, The clients were Lenovo Thinkpads with an Intel Pentium Core 2 Duo 1.8GHz CPU running Redhat Fedora 9, and Pentium 4 Redhat Fedora 8 desktop machines. We measured overall throughput in a setting where the two clients simultaneously issued transactions to the server running our ODP software, connecting to a MySQL database through MySQL Proxy [44]. As a baseline control setup we ran the same clients connected directly to the server-hosted MySQL database.

We soon discovered that in this setup the 1GBps network bandwidth is easily surpassing the processing ability of our baseline, thus we focused mainly on understanding the behavior of ODP vs. baseline MySQL as a function of network latency. To this end we modulated network latency at the kernel level using the NetEm [36] network emulation tool, which delays packets in the outgoing network queue[1]

Figure 2 shows the throughput in queries per second obtained using a remote MySQL database with no server guarantees, and the throughput obtained in our strawman ODP implementation with full privacy and correctness as-

[1] Effective bandwidth was also slightly decreased by the latency, since the TCP window sizes are fixed.

surances. We vary link latency from 0.1ms to 100ms, sampling at growing intervals to suit the log scale X axis.

From Strawman to Efficient Prototype. The strawman ODP implementation could support over 30 queries per second with full assurances. We believe this throughput can be increased by at least one order of magnitude by an industry level prototype which would consider the following bottlenecks of the strawman solution.

Multiple process forks. We used Java to manage all the communication aspects, as its pre-existing constructs reduce coding and debugging time. Additionally, a C-based Lex/Yacc parser was the most natural mechanism to detect conflicts between SQL transactions. To obtain the most functionality in the shortest amount of time, we decided to launch a new Lex/Yacc based conflict detection process from Java for every SQL statement. The result is that we incur several process forks for each processed transaction, launching both a shell and the parser once for each statement in each transaction on each client. Additionally, the conflict detection operates as a separate C-based executable. While process forks themselves are relatively cheap, incurring several in succession while the client waits for the commit creates a low performance cap. We profiled the time required to launch a shell and application at approximately 2ms – this accounts for a large portion of our overhead.

Synchronous client. The MySQL command line and stdin piping was used as our application client. This incurs the full latency of each transaction as a transaction throughput cap. Having two concurrent clients alleviates this slightly, but issuing multiple simultaneous transactions from each client would decrease the impact of latency on throughput. Additionally, part of this benefit can be received by continuing each single-threaded client before the commit has been applied – even at the risk of causing more conflicts, e.g., by creating the possibility for client conflicts with itself.

Multiple TCP connection setups. Instead of reusing client-server TCP connections, the strawman creates a new connection on each request. Multiple requests are constructed per transaction. This design choice is a result of the Java XMLRPC server we are building on; this slowdown can be eliminating by choosing a different XMLRPC implementation.

Java VM overhead. Choosing Java as the implementation language allowed quick prototyping. The disadvantage is that the Java VM (even after just-in-time compilation) can run many (I/O) tasks slower than a streamlined implementation compiled to machine byte code. MySQL, by comparision, is implemented in C.

Lua scripting overhead. The MySQL Proxy allows the capture of sessions without re-building a custom MySQL listener. This allowed fast integration with MySQL-enabled applications. The interface to MySQL proxy consists of a Lua [44] script parsed at runtime. Application logic in this Lua script runs considerably slower than it would if implemented directly.

Also, a set of costs *cannot* be eliminated due to the core nature of the protocol, including:

- Symmetric key encryption: two symmetric key encryptions, one approximately the size of the transaction description, and the fixed size commit of around 100 bytes. Additionally, the corresponding decryptions on each client.
- Hash function computations, for MACs and hash chains: the overall quantity of data hashed per transactions per client is the size of the transaction description, plus a small amount around 100 bytes.
- Two asynchronous TCP round trips: latency will not affect the throughput of asynchronous (conflict-free) clients. The network bandwidth consumed per transaction is slightly higher than in MySQL, since we send transactions back out to all clients.
- Proxy costs: organizing hash chains, ordering and transmitting incoming transactions. This program overhead however, is likely always smaller then the actual transaction application times.
- Conflict detection cost: the time required to determine if the order of two transactions affects the outcome. This is an application specific cost – a function of the conflict definition.
- Clients running each individual transaction description: This cost is incurred at the server in a MySQL-only scenario.

Ultimately, in an industry-level prototype, we estimate throughputs of roughly the same order of magnitude as an un-secured MySQL server.

8 Conclusions

In this paper, we introduced a novel paradigm for secure outsourcing of data management primitives, specifically durability and availability with assurances of data confidentiality and access privacy. We designed, implemented, and evaluated a strawman implementation that validates the feasibility of the new paradigm, running at tens of queries per second. We identified key efficiency bottlenecks that can be eliminated in an industry-level prototype to achieve orders of magnitude higher throughputs.

9 Acknowledgments

We would like to thank our anonymous reviewers, who offered helpful insights. We would also like to thank Smitha Thomas for assistance with the implementation.

References

[1] Amazon Elastic Compute Cloud. Online at http://aws.amazon.com/ec2.

[2] Amazon Simple Storage Service. Online at http://aws.amazon.com/s3.

[3] Amazon Web Services. Online at http://aws.amazon.com.

[4] Google App Engine. Online at http://code.google.com/.

[5] IBM Data Center Outsourcing Services. Online at http://www-1.ibm.com/services/.

[6] Inetu.net Managed Database Hosting. Online at http://www.inetu.net.

[7] MySQL. Online at http://www.mysql.com/.

[8] Opendb.com Web Database Hosting. Online at http://www.opendb.com.

[9] Sun Utility Computing. Online at http://www.sun.com/service/sungrid/index.jsp.

[10] Yahoo Briefcase. Online at http://briefcase.yahoo.com.

[11] Atul Adya and Barbara Liskov. Generalized isolation level definitions. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 67–78, 2000.

[12] American National Standard for Information Systems, 1819 L Street, NW, Washington, DC 20036, USA. *ANSI X3.135-1992 – Database Language SQL*, 1992.

[13] Yair Amir and Ciprian Tutu. From total order to database replication. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2000.

[14] M. Blaze. A Cryptographic File System for Unix. In *Proceedings of the first ACM Conference on Computer and Communications Security*, pages 9–16, Fairfax, VA, 1993. ACM.

[15] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[16] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Proceedings of Eurocrypt 2004*, pages 506–522. LNCS 3027, 2004.

[17] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *EuroCrypt*, 2003.

[18] R. Brinkman, J. Doumen, and W. Jonker. Using secret sharing for searching in encrypted data. In *Secure Data Management*, 2004.

[19] G. Cattaneo, L. Catuogno, A. Del Sorbo, and P. Persiano. The Design and Implementation of a Transparent Cryptographic Filesystem for UNIX. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 245–252, Boston, MA, June 2001.

[20] Y. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. Cryptology ePrint Archive, Report 2004/051, 2004. http://eprint.iacr.org/.

[21] CNN. Feds seek Google records in porn probe. Online at http://www.cnn.com, January 2006.

[22] CNN. YouTube ordered to reveal its viewers. Online at http://www.cnn.com, July 2008.

[23] Premkumar T. Devanbu, Michael Gertz, April Kwong, Chip Martel, G. Nuckolls, and Stuart G. Stubblebine. Flexible authentication of XML documents. In *ACM Conference on Computer and Communications Security*, pages 136–145, 2001.

[24] Premkumar T. Devanbu, Michael Gertz, Chip Martel, and Stuart G. Stubblebine. Authentic third-party data publication. In *IFIP Workshop on Database Security*, pages 101–112, 2000.

[25] Einar Mykletun and Maithili Narasimha and Gene Tsudik. Signature Bouquets: Immutability for Aggregated/Condensed Signatures. In *Proceedings of the European Symposium on Research in Computer Security ESORICS*, pages 160–176, 2004.

[26] Gartner, Inc. Server Storage and RAID Worldwide. Technical report, Gartner Group/Dataquest, 1999. www.gartner.com.

[27] S. Ghemawat, H. Gobioff, and S. T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing, NY, October 2003. ACM SIGOPS.

[28] E. Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. http://eprint.iacr.org/2003/216/.

[29] O. Goldreich. *Foundations of Cryptography*. Cambridge University Press, 2001.

[30] P. Golle, J. Staddon, and B. Waters. Secure conjunctive keyword search over encrypted data. In *Proceedings of ACNS*, pages 31–45. Springer-Verlag; Lecture Notes in Computer Science 3089, 2004.

[31] Philippe Golle and Ilya Mironov. Uncheatable distributed computations. In *Proceedings of the 2001 Conference on Topics in Cryptology*, pages 425–440. Springer-Verlag, 2001.

[32] P. C. Gutmann. Secure filesystem (SFS) for DOS/Windows. `www.cs.auckland.ac.nz/~pgut001/sfs/index.html`, 1994.

[33] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 216–227. ACM Press, 2002.

[34] H. Hacigumus, B. R. Iyer, and S. Mehrotra. Providing database as a service. In *IEEE International Conference on Data Engineering (ICDE)*, 2002.

[35] J. S. Heidemann and G. J. Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[36] Stephen Hemminger. Network emulation with netem (lca 2005). Online at `http://developer.osdl.org/shemminger/netem/LCA2005_paper.pdf`, April 2005.

[37] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *Proceedings of ACM SIGMOD*, 2004.

[38] HweeHwa Pang and Arpit Jain and Krithi Ramamritham and Kian-Lee Tan. Verifying Completeness of Relational Query Results in Data Publishing. In *Proceedings of ACM SIGMOD*, 2005.

[39] Jetico, Inc. BestCrypt software home page. `www.jetico.com`, 2002.

[40] A. Kashyap, S. Patil, G. Sivathanu, and E. Zadok. I3FS: An In-Kernel Integrity Checker and Intrusion Detection File System. In *Proceedings of the 18th USENIX Large Installation System Administration Conference (LISA 2004)*, pages 69–79, Atlanta, GA, November 2004. USENIX Association.

[41] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*, pages 134–143, 2000.

[42] G. Kim and E. Spafford. Experiences with Tripwire: Using Integrity Checkers for Intrusion Detection. In *Proceedings of the Usenix System Administration, Networking and Security (SANS III)*, 1994.

[43] G. Kim and E. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *Proceedings of the 2nd ACM Conference on Computer Commuications and Society (CCS)*, November 1994.

[44] Jan Kneschke, Lenz Grimmer, Martin Brown, Giuseppe Maxia, and Kay Röpke. Mysql proxy - mysql forge wiki. Online at `http://forge.mysql.com/wiki/MySQL_Proxy`, 2008.

[45] Kyriacos Pavlou and Richard T. Snodgrass. Forensic Analysis of Database Tampering. In *Proceedings of ACM SIGMOD*, 2006.

[46] M. Sullivan and M. Stonebraker. Using Write Protected Data Structures to Improve Software Fault Tolerance in Highly Available Database Management Systems. In *Proceedings of VLDB*, 1991.

[47] Maithili Narasimha and Gene Tsudik. DSAC: integrity for outsourced databases with signature aggregation and chaining. Technical report, 2005.

[48] Maithili Narasimha and Gene Tsudik. Authentication of Outsourced Databases using Signature Aggregation and Chaining. In *Proceedings of DASFAA*, 2006.

[49] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. Stubblebine. A general model for authenticated data structures. Technical report, 2001.

[50] Charles Martel, Glen Nuckolls, Premkumar Devanbu, Michael Gertz, April Kwong, and Stuart G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[51] David Mazières and Dennis Shasha. Building secure file systems out of byzantine storage. In *PODC '02: Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, pages 108–117, New York, NY, USA, 2002. ACM Press.

[52] A. D. McDonald and M. G. Kuhn. StegFS: A Steganographic File System for Linux. In *Information Hiding*, pages 462–477, 1999.

[53] R. Merkle. Protocols for public key cryptosystems. In *IEEE Symposium on Research in Security and Privacy*, 1980.

[54] Microsoft Research. Encrypting File System for Windows 2000. Technical report, Microsoft Corporation, July 1999. www.microsoft.com/windows2000/techinfo/howitworks/security/encrypt.asp.

[55] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. In *ISOC Symposium on Network and Distributed Systems Security NDSS*, 2004.

[56] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. In *Proceedings of Network and Distributed System Security (NDSS)*, 2004.

[57] E. Mykletun, M. Narasimha, and G. Tsudik. Signature bouquets: Immutability for aggregated/condensed signatures. In *Computer Security - ESORICS 2004*, volume 3193 of *Lecture Notes in Computer Science*, pages 160–176. Springer, 2004.

[58] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of EuroCrypt*, 1999.

[59] Pascal Paillier. A trapdoor permutation equivalent to factoring. In *PKC '99: Proceedings of the Second International Workshop on Practice and Theory in Public Key Cryptography*, pages 219–222, London, UK, 1999. Springer-Verlag.

[60] HweeHwa Pang and Kian-Lee Tan. Authenticating query results in edge computing. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, page 560, Washington, DC, USA, 2004. IEEE Computer Society.

[61] Philip Bohannon and Rajeev Rastogi and S. Seshadri and Avi Silberschatz and S. Sudarshan. Using Codewords to Protect Database Data from a Class of Software Errors. In *Proceedings of ICDE*, 1999.

[62] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 89–101, Monterey, CA, January 2002. USENIX Association.

[63] Richard T. Snodgrass and Stanley Yao and Christian Collberg. Tamper Detection in Audit Logs. In *Proceedings of VLDB*, 2004.

[64] Radu Sion. Query execution assurance for outsourced databases. In *Proceedings of the Very Large Databases Conference VLDB*, 2005.

[65] G. Sivathanu, C. P. Wright, and E. Zadok. Enhancing File System Integrity Through Checksums. Technical Report FSL-04-04, Computer Science Department, Stony Brook University, May 2004. www.fsl.cs.sunysb.edu/docs/nc-checksum-tr/nc-checksum.pdf.

[66] D. Xiaodong Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy (S&P 2000)*. IEEE Computer Society, 2000.

[67] Tingjian Ge and Stan Zdonik. Answering aggregation queries in a secure system model. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 519–530. VLDB Endowment, 2007.

[68] Peter Williams and Radu Sion. Usable PIR. In *Proceedings of the 2008 Network and Distributed System Security (NDSS) Symposium*, 2008.

[69] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A Secure and Convenient Cryptographic File System. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210, San Antonio, TX, June 2003. USENIX Association.